1. Gaussian quadrature rules

By hand, derive the 3-point Gauss-Legendre quadrature rule by guessing the symmetry and requiring that it has polynomial degree 5.

Answer in Wikipedia for reference: ↳

Number of points, n	Points, x_i		Weights, w_i	
1	0		2	
2	$\pm\frac{1}{\sqrt{3}}$	±0.57735...	1	
3	0		$\frac{8}{9}$	0.888889...
	$\pm\sqrt{\frac{3}{5}}$	±0.774597...	$\frac{5}{9}$	0.555556...

We expect the symmetry $(x_0, x_1, x_2) = (-r, 0, r)$ for some $r \in (0,1]$, and $\omega_2 = \omega_0$. Thus $Q(f) = \omega_0 f(-r) + \omega_1 f(0) + \omega_0 f(r)$.
(If this guess is wrong, we will know about it as the calculation proceeds.)
Want poly degree 5, that is $Q(f) = \int_{-1}^{1} f \quad \forall f \in \{1, \cdot, \cdot^2, \cdot^3, \cdot^4, \cdot^5\}$.

(0) $f(x) = 1$

$$\omega_0 + \omega_1 + \omega_0 = \int_{-1}^{1} 1 dx = 2 \quad : \quad 2\omega_0 + \omega_1 = 2 \quad (1)$$

(1) $f(x) = x$

$$\omega_0(-r) + \omega_1(0) + \omega_0(r) = \int_{-1}^{1} x dx = 0 \ : \ 0 = 0.$$

Accomplished by the symmetry.

(2) $f(x) = x^2$

$$\omega_0(r)^2 + \omega_1(0)^2 + \omega_0 r^2 = \int_{-1}^{1} x^2 dx = \frac{2}{3} \ : \quad 2\omega_0 r^2 = \frac{2}{3}, \quad \omega_0 r^2 = \frac{1}{3} \quad (2)$$

(3) $f(x) = x^3$

$$\omega_0(-r)^3 + \omega_1(0)^3 + \omega_0 r^3 = \int_{-1}^{1} x^3 dx = 0 \ : \ 0 = 0.$$

Accomplished by the symmetry.

(4) $f(x) = x^4$

$$\omega_0(-r)^4 + \omega_1(0)^4 + \omega_0 r^4 = \int_{-1}^{1} x^4 dx = \frac{2}{5} \ : \quad 2\omega_0 r^4 = \frac{2}{5}, \quad \omega_0 r^4 = \frac{1}{5} \quad (3)$$

(5) $f(x) = x^5$. Again by symmetry.

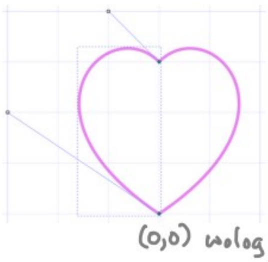
So we have 3 (nonlinear) equations in the 3 unknowns ω_0, ω_1, r:
Dividing (3) by (2), we have $r^2 = \frac{\frac{1}{5}}{\frac{1}{3}} = \frac{3}{5}$, so $r = \sqrt{\frac{3}{5}}$.

Then (2) gives $\omega_0 \cdot \frac{3}{5} = \frac{1}{3}$, so $\omega_0 = \frac{5}{9} = \omega_2$.

Finally (1) gives $\omega_1 = 2 - 2\left(\frac{5}{9}\right) = \frac{18-10}{9}$, $\omega_1 = \frac{8}{9}$.
Summarizing, $Q(f) = \frac{5}{9} f\left(-\sqrt{\frac{3}{5}}\right) + \frac{8}{9} f(0) + \frac{5}{9} f\left(\sqrt{\frac{3}{5}}\right)$.

2. Using a quadrature rule

Obtain an accurate approximation to the length of this curve made of two Bezier segments, if the grid squares have side 1 meter.



The left half of the curve is a Bézier segment $P(t)$ with
$P_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $P_1 = \begin{bmatrix} -3 \\ 2 \end{bmatrix}$, $P_2 = \begin{bmatrix} -1 \\ 4 \end{bmatrix}$, $P_3 = \begin{bmatrix} 0 \\ 3 \end{bmatrix}$.

The length of the segment is the integral of the speed

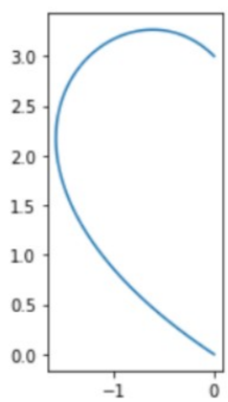
$$L = \int_0^1 \text{speed}(t)\,dt = \int_0^1 \|P'(t)\|_2\,dt.$$

So we just need to differentiate the formula $P(t)$ and integrate its 2-norm.

```python
from nsm import *

def bezier(P,t):  # columns of P are P0, P1, P2, P3
    P0,P1,P2,P3 = P.T
    s = 1 - t
    x =  s**3*P0[0] + 3*s**2*t*P1[0] + 3*s*t**2*P2[0] + t**3*P3[0]
    y =  s**3*P0[1] + 3*s**2*t*P1[1] + 3*s*t**2*P2[1] + t**3*P3[1]
    return x,y

P = np.array([[0,-3,-1,0],
              [0, 2, 4,3]]) # columns of P are P0, P1, P2, P3
```

check curve looks correct

```python
t = np.linspace(0,1,100)
plt.subplot(111,aspect=1)
plt.plot(*bezier(P,t));
```



compute the speed:

```python
t = sp.symbols('t')
curvex,curvey = bezier(P,t)
print('the curve:')
display(curvex.expand())
display(curvey.expand())
print('the velocity:')
vx = curvex.expand().diff(t)
vy = curvey.expand().diff(t)
display(vx)
display(vy)
print('the speed:')
speed = sp.sqrt((vx**2 + vy**2).expand())
display(speed)
```

the curve:

$-6t^3 + 15t^2 - 9t$

$-3t^3 + 6t$

the velocity:

$-18t^2 + 30t - 9$

$6 - 9t^2$

the speed:

$\sqrt{405t^4 - 1080t^3 + 1116t^2 - 540t + 117}$

```python
speedfunc = sp.lambdify(t,speed,'numpy')
# check speedfunc
print(speedfunc(np.array([0,1])))
```

```
[10.81665383  4.24264069]
```

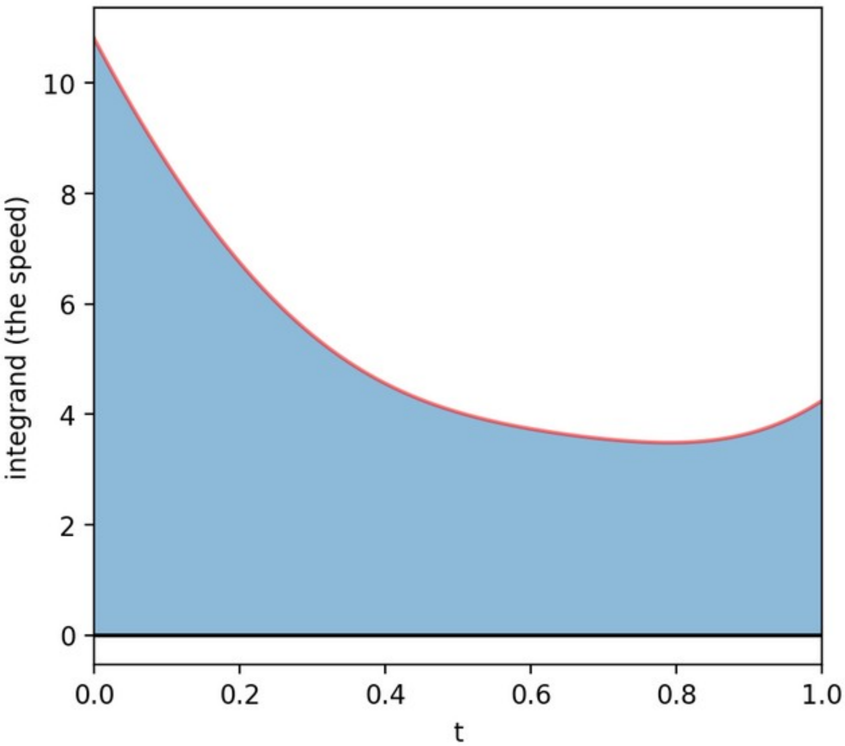
Accurately approximate length by numerical quadrature

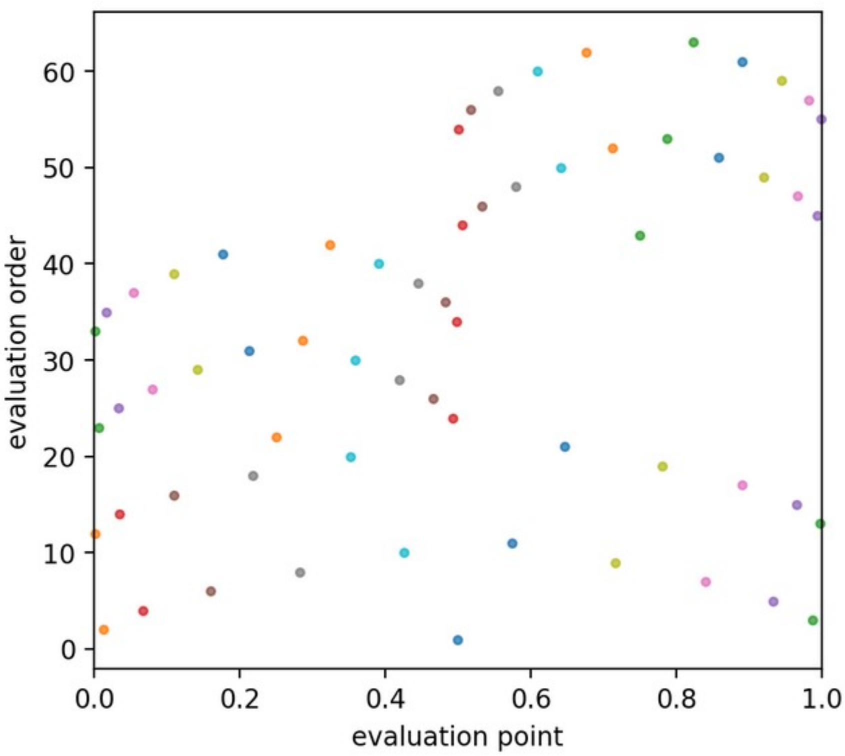
Before running this, as a sanity check, I'm eyeballing the curve and estimating the length to be about 5.

```
import numpy as np
import sympy as sp
import pylab as plt
%config InlineBackend.figure_format = 'retina'
```

```
# plot the velocity
def vfunc(t):
    global nevals,ncalls,DIAGNOSTICS
    if DIAGNOSTICS:
        ncalls += 1
        try:
            nevals += len(t)
            print(len(t),'values requested')
            plt.plot(t,[ncalls]*len(t),'o',ms=3,alpha=.75,clip_on=False)
        except: # t a scalar
            nevals += 1
            plt.plot( t, ncalls, 'o',ms=3,alpha=0.75,clip_on=False)

    return np.sqrt( (-18*t**2 + 30*t - 9)**2 + (6 - 9*t**2)**2 )  # the speed

t = np.linspace(0,1,500)
DIAGNOSTICS = False
v = vfunc(t)
plt.figure(figsize=(5,10))
plt.subplot(211)
plt.fill_between(t,v,alpha=.5);
plt.plot        (t,v,alpha=.5,color='r');
plt.axhline(0,color='k')
plt.ylabel('integrand (the speed)'); plt.xlabel('t')
plt.xlim(0,1)

plt.subplot(212)
from scipy.integrate import quad
nevals = 0
ncalls = 0
DIAGNOSTICS = True
I,err = quad( vfunc,0,1)
plt.xlabel('evaluation point')
plt.ylabel('evaluation order')
plt.xlim(0,1)
print('integral estimate:',I)
print('error estimate:',err)
print('number of function evaluations:',nevals)
```

```
integral estimate: 5.103195629014709
error estimate: 7.103803580220992e-13
number of function evaluations: 63
```





The routine quad is evidently using adaptive quadrature with an (11-point, 21-point) rule pair. Googling suggests this is a Gauss-Kronrod pair. A single subdivision was found sufficient to satisfy the default error tolerance.

For the full "heart" curve, we obtain

```
np.round(2*I,11)
```

```
10.20639125803
```

rounding to 13 digits because these, and only these, are expected to be accurate, based on the error estimate.

3 (a) $F\left(\begin{bmatrix}u\\v\end{bmatrix}\right) = \begin{bmatrix}u^3-v\\u^2+v^2-1\end{bmatrix}$ $\qquad F\left(\begin{bmatrix}1\\2\end{bmatrix}\right) = \begin{bmatrix}1^3-2\\1^2+2^2-1\end{bmatrix} = \begin{bmatrix}-1\\4\end{bmatrix}$

$F'\left(\begin{bmatrix}u\\v\end{bmatrix}\right) = \begin{bmatrix}3u^2 & -1\\2u & 2v\end{bmatrix}$ $\qquad F'\left(\begin{bmatrix}1\\2\end{bmatrix}\right) = \begin{bmatrix}3\cdot1^2 & -1\\2\cdot1 & 2\cdot2\end{bmatrix} = \begin{bmatrix}3 & -1\\2 & 4\end{bmatrix}$

1st Newton step $\quad s^{(0)}$ determined by $F'(x^{(0)})\,s^{(0)} = -F(x^{(0)})$.

Writing $s^{(0)} = \begin{bmatrix}\delta u\\\delta v\end{bmatrix}$, that is $\begin{bmatrix}3 & -1\\2 & 4\end{bmatrix}\begin{bmatrix}\delta u\\\delta v\end{bmatrix} = \begin{bmatrix}+1\\-4\end{bmatrix}$

$\left[\begin{array}{cc|c}3 & -1 & 1\\2 & 4 & -4\end{array}\right] \xrightarrow{②+4①} \left[\begin{array}{cc|c}3 & -1 & 1\\14 & 0 & 0\end{array}\right] \longrightarrow 14\delta u = 0,\ \boxed{\delta u = 0}$

Then ② $\rightarrow$ $0 - 1\delta v = 1 \rightarrow \boxed{\delta v = -1}$

$\left.\right\}\ s^{(0)} = \begin{bmatrix}0\\-1\end{bmatrix}$

Thus $x^{(1)} = x^{(0)} + s^{(0)} = \begin{bmatrix}1\\2\end{bmatrix} + \begin{bmatrix}0\\-1\end{bmatrix} = \begin{bmatrix}1\\1\end{bmatrix} = x^{(1)}$

(b) Broyden update of Jacobian.

Formula is $\quad B^{(k+1)} = B^{(k)} + \dfrac{(\Delta^{(k)} - B^{(k)}s^{(k)})}{s^{(k)T}s^{(k)}}\, s^{(k)T}$

where $B^{(k)}$ is approx to Jacobian at $x^{(k)}$, $\Delta^{(k)}$ is $F(x^{(k+1)}) - F(x^{(k)})$.

Here $B^{(0)} = F'\left(\begin{bmatrix}1\\2\end{bmatrix}\right) = \begin{bmatrix}3 & -1\\2 & 4\end{bmatrix}$

$\Delta^{(0)} = F(x^{(1)}) - F(x^{(0)}) = F\left(\begin{bmatrix}1\\1\end{bmatrix}\right) - F\left(\begin{bmatrix}1\\2\end{bmatrix}\right) = \begin{bmatrix}1^3-1\\1^2+1^2-1\end{bmatrix} - \begin{bmatrix}-1\\4\end{bmatrix} = \begin{bmatrix}0\\1\end{bmatrix} - \begin{bmatrix}-1\\4\end{bmatrix} = \begin{bmatrix}1\\-3\end{bmatrix}$

$s^{(0)} = \begin{bmatrix}0\\-1\end{bmatrix}$, $\quad s^{(0)T} = [0, -1]$, $\quad s^{(0)T}s^{(0)} = [0,-1]\begin{bmatrix}0\\-1\end{bmatrix} = 1$.

So

$B^{(1)} = \begin{bmatrix}3 & -1\\2 & 4\end{bmatrix} + \left(\begin{bmatrix}1\\-3\end{bmatrix} - \begin{bmatrix}3 & -1\\2 & 4\end{bmatrix}\begin{bmatrix}0\\-1\end{bmatrix}\right)[0,-1]$

$= \begin{bmatrix}3 & -1\\2 & 4\end{bmatrix} + \left(\begin{bmatrix}1\\-3\end{bmatrix} - \begin{bmatrix}1\\-4\end{bmatrix}\right)[0,-1] = \begin{bmatrix}3 & -1\\2 & 4\end{bmatrix} + \begin{bmatrix}0\\1\end{bmatrix}[0,-1] = \begin{bmatrix}3 & -1\\2 & 4\end{bmatrix} + \begin{bmatrix}0 & 0\\0 & -1\end{bmatrix}$

$= \begin{bmatrix}3 & -1\\2 & 3\end{bmatrix}$.

The actual Jacobian at $x^{(1)} = \begin{bmatrix}1\\1\end{bmatrix}$ is $F'\left(\begin{bmatrix}1\\1\end{bmatrix}\right) = \begin{bmatrix}3\cdot1^2 & -1\\2\cdot1 & 2\cdot1\end{bmatrix} = \begin{bmatrix}3 & -1\\2 & 2\end{bmatrix}$.

The approximation $B^{(1)}$ is not great. We can attribute this to the large size of the step $s^{(1)}$ so that the secant approximation to the derivative is poor. However, it makes sense that only the 2nd column was altered because it represents $\partial/\partial v$ and only v changed in step 1.

I'll check my arithmetic with Python:

```python
from nsm import *

def arr(*x): return np.array(x)  # to make it easier to create arrays

def F(x):
    u,v = x
    return arr( u**3 - v, u**2 + v**2 - 1 )

def Fp(x):
    u,v = x
    return arr( arr(3*u**2,-1), arr(2*u,2*v) )

x0 = arr(1,2)
F0 = F(x0)
Fp0 = Fp(x0)
print('x0',x0)
print('F0',F0)
print('Fp0',Fp0)
```

```
x0 [1 2]
F0 [-1  4]
Fp0 [[ 3 -1]
 [ 2  4]]
```

```python
s0 = np.linalg.solve(Fp0,-F0)
s0
print('s0',s0)
x1 = x0 + s0
print('x1',x1)
F1 = F(x1)
Fp1 = Fp(x1)
print('F1',F1)
print('Fp1',Fp1)
```

```
s0 [ 0. -1.]
x1 [1. 1.]
F1 [0. 1.]
Fp1 [[ 3. -1.]
 [ 2.  2.]]
```

```python
# Broyden update of Fp0
B0 = Fp0
Delta0 = F1-F0
print('Delta0',Delta0)
B1 = Fp0 + np.outer(Delta0 - B0@s0,s0)/np.dot(s0,s0)
print('B1',B1)
```

```
Delta0 [ 1. -3.]
B1 [[ 3. -1.]
 [ 2.  3.]]
```

Everything agrees with hand-calculation.

Let's try to solve the system using only Broyden updating

```python
x = x0
B = Fp(x)
tol = 1.e-8
nsteps = 0
while True:
    s = np.linalg.solve(B,-F(x))
    newx = x + s
    newF = F(newx)
    Delta = newF - F(x)
    B += np.outer(Delta - B@s , s )/np.dot(s,s)
    x = newx
    nsteps += 1
    if np.linalg.norm(s) < tol:
        break
print('Approximate solution',x,'in',nsteps,'steps')
```

```
Approximate solution [0.82603136 0.56362416] in 9 steps
```

4. Sherman-Morrison update

(a) Validate formula

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + (v^TA^{-1}u)}$$

(assuming denominator $\neq 0$, which will be true for all sufficiently small u, v).

First let's note that for convenience we tend to be a bit sloppy about the distinction between 1-by-1 matrices and scalars. For example, in the denominator $1 + v^TA^{-1}u$, 1 is a scalar, while strictly $v^TA^{-1}u$ is a 1-by-1 matrix. For this question let's use $(v^TA^{-1}u)$ in parentheses to mean the (one) scalar entry of the 1-by-1 matrix $v^TA^{-1}u$.

For the calculation below, let's also note that $Mv^TA^{-1}uN = (v^TA^{-1}u)MN$ call it $(*)$ for matrices M, N with shapes for which the products are valid.

Compute product of alleged inverse (RHS) with $A + uv^T$

$$\text{RHS} \cdot (A + uv^T) = \left(A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}\right)(A + uv^T)$$

$$= AA^{-1} - \frac{A^{-1}uv^TA^{-1}A}{1 + (v^TA^{-1}u)} + A^{-1}uv^T - \frac{A^{-1}uv^TA^{-1}uv^T}{1 + (v^TA^{-1}u)}$$

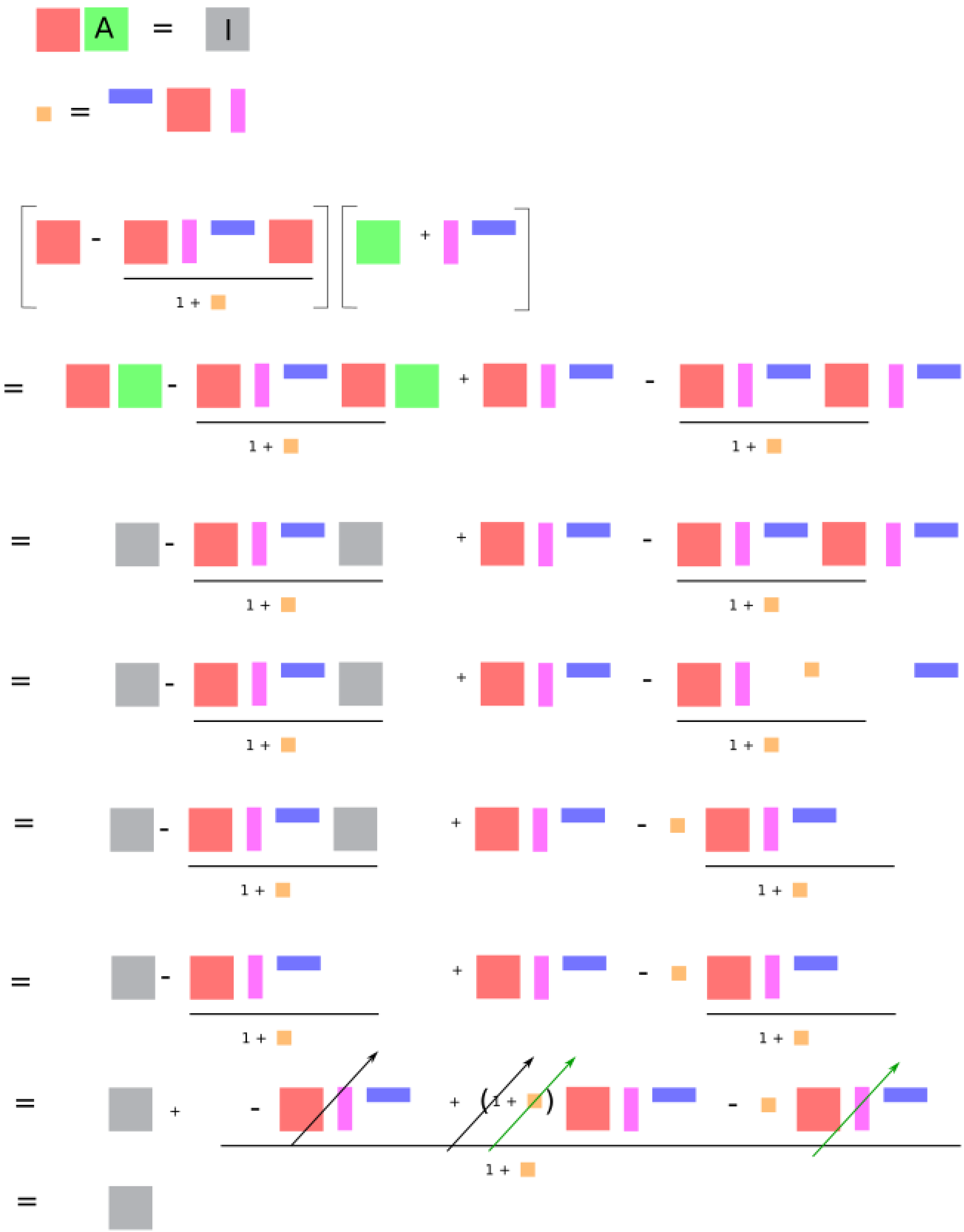
$$= I + \frac{\cancel{-A^{-1}uv^T} + \cancel{A^{-1}uv^T} + (v^TA^{-1}u)A^{-1}uv^T - A^{-1}uv^TA^{-1}uv^T}{1 + (v^TA^{-1}u)}$$

$$= I + \frac{\overbrace{(v^TA^{-1}u)}^{\text{scalar}}A^{-1}uv^T - A^{-1}u\overbrace{v^TA^{-1}u}^{\text{1-by-1 matrix}}v^T}{1 + (v^TA^{-1}u)}$$

$$= I + \frac{(v^TA^{-1}u)A^{-1}uv^T - (v^TA^{-1}u)A^{-1}uv^T}{1 + (v^TA^{-1}u)}$$ by identity $(*)$

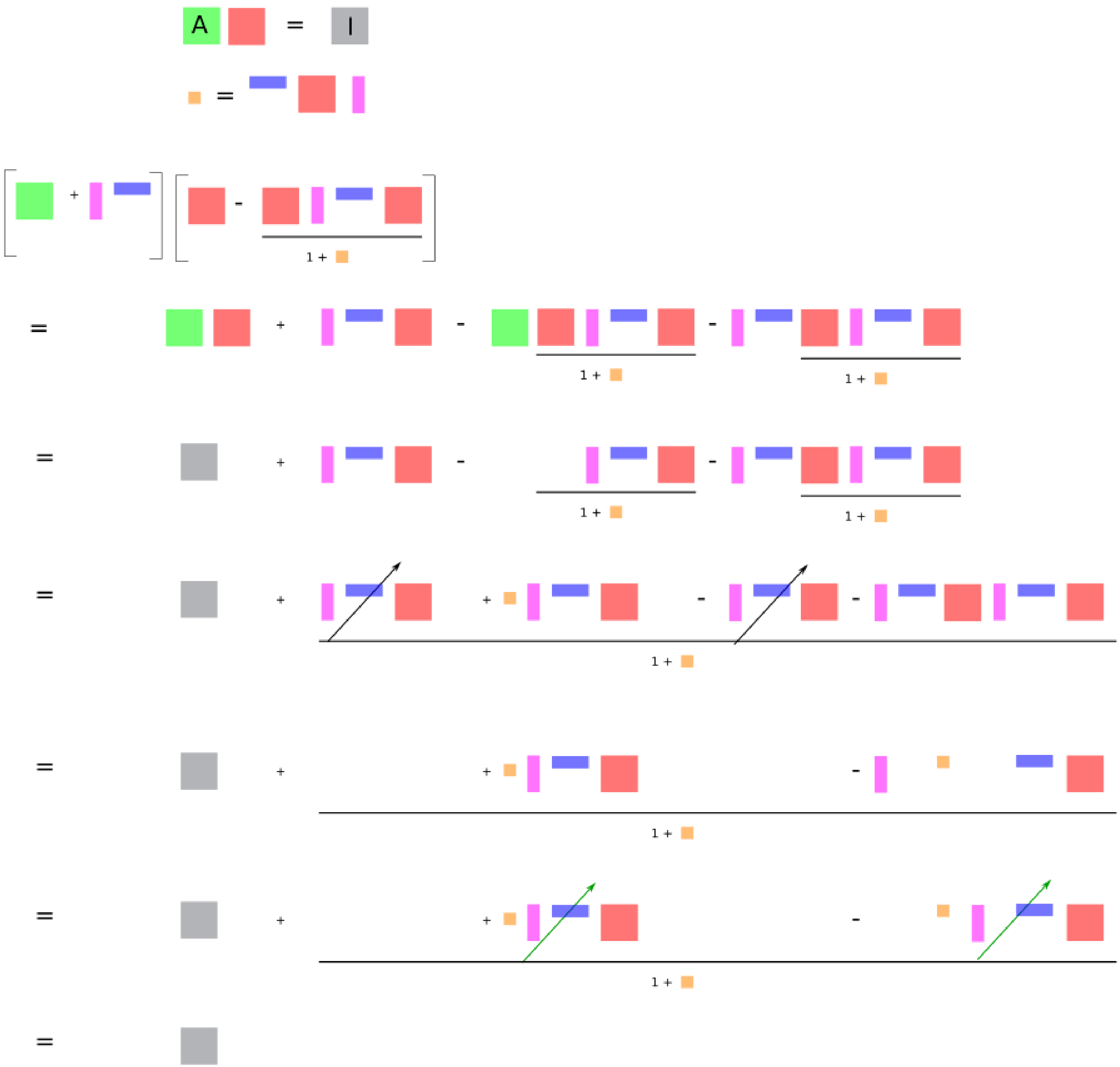
$$= I + \frac{0}{1 + v^TA^{-1}u} = I.$$ QED ☺

Pictorially,

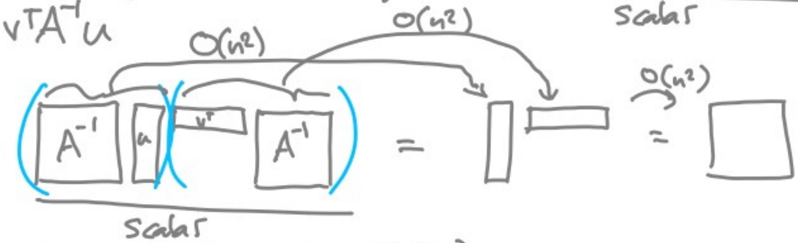


where a 1x1 matrix on the left means scalar multiplication by its one element.

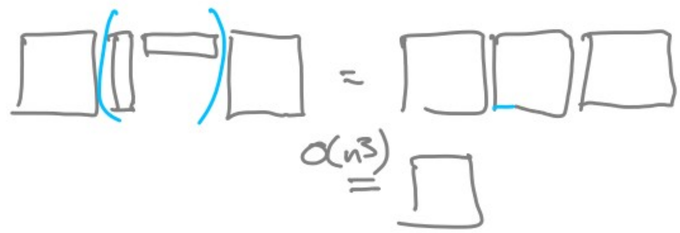
or, in the reverse order



(b) The update $\dfrac{A^{-1}uv^TA^{-1}}{1+v^TA^{-1}u}$ has the form 

If we compute it as 

we are not doing anything that costs $O(n^3)$ ops.

(If we did it like this, though, 

it would cost $O(n^3)$.)